Implementing Scrambling in Korean: A Principles and Parameters Approach

by

Franklin S. Cho

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

Author .....
Department of Electrical Engineering and Computer Science
May 26, 1995

Certified by .....
Robert C. Berwick
Professor of Computer Science and Engineering and Computational Linguistics
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Theses

Implementing Scrambling in Korean: A Principles and Parameters Approach

by

Franklin S. Cho

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

This paper describes how the most complete recent linguistic results on Korean scrambling (switching of word order) can be readily incorporated into an existing principles-and-parameters parser with minimal additional machinery. Out of all 29 sets of examples in chapters 2.2 and 3.2 of perhaps the most advanced linguistic analysis of Korean scrambling, [5], 26 sets of examples can be correctly parsed, greatly extending the variety of scrambling handled by any current parser. This approach is compared to other current approaches to scrambling, such as PRINCIPAR and Tree Adjoining Grammar.

Thesis Supervisor: Robert C. Berwick
Title: Professor of Computer Science and Engineering and Computational Linguistics

Acknowledgments

Firstly, the author would like to thank his family and friends for all their support over the years. The author also would like to thank Prof. Robert C. Berwick, Dr. Sandiway Fong, Dr. Youngsuk Lee and Dr. Dekang Lin for their generous help.

To My Parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scrambling is a complex yet common phenomenon in Korean that allows the apparent movement of a noun phrase over both short and long distances:

- Scrambling of more than one noun phrase that belongs to the same verb's argument structure, or "multiple scrambling". For example, in (1)(ii), "chayk" (book) moves in front of "Youlee", or even to the front of the sentence, as in (1)(iv).

(1) (i) Sunhee-ka Youlee-eykey [chayk hankwen]-ul senmwulhayssta
Sunhee-nom Youlee-dat [book one-volume]-acc gave-a-present
"Sunhee gave Youlee a book as a present."

(ii) sunhee-ka [chayk hankwen]-ul youlee-eykey senmwulhayssta
Sunhee-nom [book one-volume]-acc Youlee-dat gave-a-present

(iii) youlee-eykey sunhee-ka [chayk hankwen]-ul senmwulhayssta

(iv) youlee-eykey [chayk hankwen]-ul sunhee-ka senmwulhayssta

(v) [chayk hankwen]-ul sunhee-ka youlee-eykey senmwulhayssta

(vi) [chayk hankwen]-ul youlee-eykey sunhee-ka senmwulhayssta[1]

- No limit to the number of clauses that a scrambled element can cross, or "unbounded dependency," For example, in (2)(ii), "chayk" (book) can be arbitrarily

[1] [5]. example 7. See figure 1 in the text.

far from its canonical argument position:[2]

(2) (i) John-i [Mary-ka [Sally-ka Bill-eykey chayk-ul cwuessta-ko]
John-nom [Mary-nom [Sally-nom Bill-dat book-acc gave-compl]
malhayssta-ko] sayngkakhanta
said-compl] think

"John thinks that Mary said that Sally gave Bill a book."

(ii) chayk-ul$_i$ [John-i [Mary-ka [Sally-ka Bill-eykey t_i cwuessta-ko]
book-acc [John-nom [Mary-nom [Sally-nom Bill-dat gave-compl]
malhayssta-ko] sayngkakhanta]
said-compl] think]

Handling scrambling correctly is very difficult for a parser. The reason is that adding a permutation component to generate all possible word orders independently of other grammatical constraints is easy. What is more difficult is to make scrambling interact correctly with all the *other* components of the grammar, for instance those that establish the interaction of scrambling with coreference. Consider these examples:

(3) (i) $*$ Younghee-ka **ku**-eykey [**Minswu**-uy sacin]-ul poyecwuessta
Younghee-nom him-dat Minswu-gen picture-acc showed

'Younghee showed **him Minswu's** picture'

(ii) Younghee-ka [**Minswu**-uy sacin]$_i$-ul **ku**-eykey t_i poyecwuessta[3]

(iii) [**Minswu**-uy sacin]$_i$-ul Younghee-ka **ku**-eykey t_i poyecwuessta[4]

The coreference relation is indicated by bold face, and the filler-gap relation (or, equivalently, antecedent-trace relation) by coindexation. (3) shows that scrambling interacts with coreference: the scrambled versions (3)(ii) and (3)(iii) are acceptable, but the canonical version (3)(i) is not. So, scrambling "saves" a sentence in (3). Now, consider these examples:

---

[2][5], p. 5

[3]Here, "t_i" denotes a link between the canonical argument position for "chayk"(book) and its actual position in the sentence. At this point, I remain theory-neutral as to the exact nature of "t_i". It could be implemented in several ways.

[4][5], example 81.

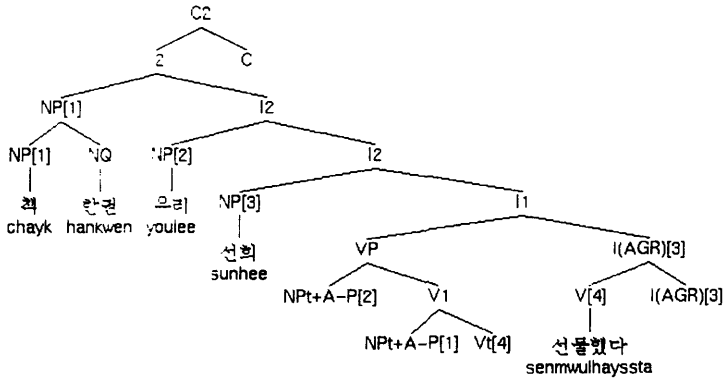

Figure 1-1: A Parsed Korean Example Sentence 1(vi)

(4) (i) [**Minswu**-uy tongsayng]-i **ku**-eykey sacin-ul poyecwuessta
Minswu-gen brother-nom him-dat picture-acc showed
'**Minswu**'s brother showed **him** a picture.'

(ii) $*$**ku**-eykey [**Minswu**-uy tongsayng]-i sacin-ul poyecwuessta[5]

Conversely, the canonical version (4)(i) is acceptable but the scrambled version (4)(ii) is not. So, scrambling "destroys" a sentence in (4). Therefore, scrambling is much more complicated than simply generating correct word orders. In both (3) and (4), scrambling interacts with a coreference constraint referred to as "Condition C" in the linguistic literature. Roughly, Condition C states that a referring expression e.g., "John," "the house," must not be bound anywhere in a sentence.[6] (4)(ii) is ruled out by Condition C, since "Minswu," a referring expression, is bound by "ku." Such an interaction must be taken into account when parsing these examples. As far as it is known, the system presented here is the first that can correctly parse such a wide range of scrambling examples. Although no good quantitative measures are known to us, scrambling seems quite common in Korean, and is therefore important for parsing. Figure 1-1 shows an example of the parser's actual output on a scrambled example

[5][5], example 82.

[6]A node A binds another node B iff A and B are coindexed, and A c-commands B. A c-commands B iff the lowest branching node which dominates A also dominates B. For example, in [α [...[β θ]...]], α c-commands β and θ. This is the canonical definition of binding, and this definition will be modified later, as shown in Figure 2-3.

sentence.

Chapter 2

Implementation

2.1 A Simple Scrambling Mechanism

Let us see how to parse scrambled sentences using a constraint-based parser, **Pappi** [2]. As a first approximation, here is a simplified description of the scrambling mechanism (Figure 2-1), showing only the modules relevant to the scrambling-coreference relations. In subsequent sections, I will refine this analysis to accomodate new examples. (There are many more filters and generators in **Pappi** than the figure shows.)

The key idea behind constraint-based parsing is to reproduce complex surface sentence patterns by the interaction of separable but linked "modules," each handling a different kind of constraint. For instance, our examples (1)-(4) motivate four modules:

1. One to move NP's from their canonical locations.
2. One to coindex filler NP's with their gaps and other NP's.
3. One to check whether this indexing meets Condition C.
4. One to move variables into proper scope (assuming a typed first-order predicate calculus (FOPC) representation.)

First, a LR(1)-based bottom-up shift-reduce parser is used to recover the phrase structure. Note that this parser allows an NP to freely attach to the beginning of a

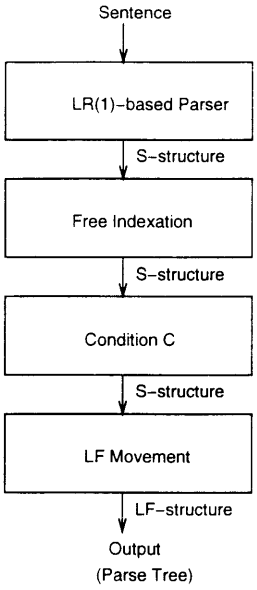

Figure 2-1: A Simple Scrambling Mechanism

sentence (or, equivalently, adjoin to an Inflection Phrase(IP)) or to the beginning of a verb phrase (or, equivalently, adjoin to a verb phrase), to account for scrambling.[1]

A mechanism such as the "Free Indexation" mechanism is called a "generator." A generator generates new structures based on the old structure that was passed to it. For example, the "Free Indexation" generator takes a parse tree without indices, and generates parse trees with indices assigned to each NP and trace (or, equivalently, gap). This generator generates all possible coindexations between the NP's and their gaps, and among the NP's.[3]

A mechanism such as the "Condition C" mechanism is called a "filter." A filter eliminates the wrong structures that enter it, and pass the correct structures to the next filter or generator. For example, the "Condition C" filter filters out every parse tree that violates the Condition C.

The "Logical Form (LF) Movement" mechanism performs two operations: first, it

[1]Also, the mechanism used to avoid "string vacuous scrambling" in Japanese, as described in [3] is used for Korean as well.[2] A "non-vacuous" or "visible" scrambling is a scrambling that "passes over" one or more overt elements [3].

[3]Please refer to [2] for the details on how the free indexation is implemented.

raises each quantifier (e.g., "every boy", "somebody") and attaches it to the beginning of the innermost sentence (or, equivalently, adjoins it to the nearest IP.) Then, it raises each wh-word (e.g., "who", "where") to the specifier position of the nearest Complementizer Phrase(CP).[4]

To summarize, after the phrase structure is recovered by the LR parser, these structures are passed through a series of filters and generators, until only the correct parses remain. The implementations are taken care of by following a generate and test paradigm (but in a more sophisticated way).

2.2 Subject Binding Generalization

However, this simple first order approximation does not suffice to cover all examples in Korean. Consider these examples:

(5) (i) *__ku__-ka [__Minswu__-uy emma]-lul coahanta
he-nom Minswu-gen mother-acc like
'__He__ likes __Minswu's__ mother.'

(ii) *[__Minswu__-uy emma]$_i$-lul __ku__-ka t_i coahanta[5]

(5)(i) is ruled out by Condition C, since "ku" binds "Minswu", a referring expression. (Please refer to Figure 2-2 for the definition of "binding."[6]) However, Condition C alone cannot explain why (5)(ii) is unacceptable, since nothing seems to bind "Minswu" (it is therefore free, unbound, and satisfies Condition C.) We can repair this problem by introducing a new definition of binding (defined in [4].) According to Lee, (5)(ii) is ruled out by the Subject Binding Generalization:

[4]Under the Government and Binding framework, the Complementizer Phrase (CP) immediately dominates the Inflection Phrase (or, equivalently, the Sentence), and wh-words move to the specifier position of a CP at Logical Form (LF) level. The specifier position is immediately dominated by the CP.

[5][5], example 86

[6]A node A c-commands another node B iff the lowest branching node which dominates A also dominates B. For example, in [α [...[β θ]...]], α c-commands β and θ.

(6) **Subject Binding Generalization:** If X in subject position binds Y at Deep Structure (D-structure or. equivalently, canonical predicate-argument structure)[7], then X binds Y at all levels of representation (i.e., FOPC level and the surface level).

In (5)(ii), "ku" binds "Minswu" in D-structure, and therefore "ku" still binds "Minswu" in surface structure (S-structure.) Therefore, (5)(ii) is ruled out by Condition C, since "Minswu" is bound.

To implement this, I revised the definitions of binding in the parser. The original and the new definition of binding are shown below as flowcharts.

Here is the original definition of binding (Figure 2-2):

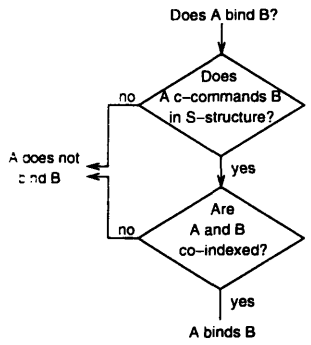

Figure 2-2: The Original Definition of Binding

[7]Each scrambled element moves back to its canonical position at D-structure.

Here is the new definition of binding (Figure 2-3):

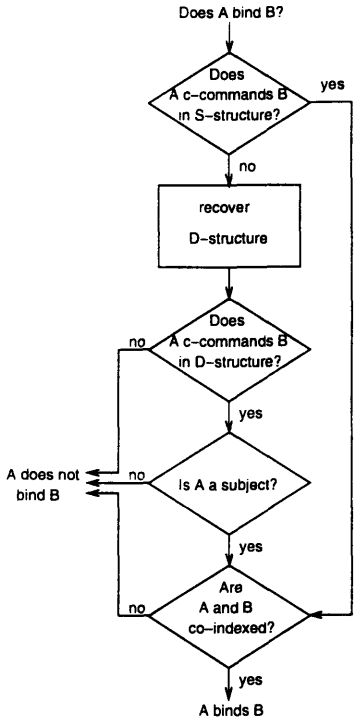

Figure 2-3: The New Definition of Binding

Notice that a change in the definition of binding automatically changes the definition of Condition C (since the code implementing Condition C calls the code that implements binding.) Also notice that the "subject" is defined as an NP with an "agent" thematic role (θ-role), following Lee's analysis.[8]

Here is how the D-structure is recovered from the S-structure: (1) recurse down the parse tree and replace each element by its D-structure element. For example, a head of a chain (or, equivalently, a filler) would be replaced by an empty element, and a trace (gap) would be replace by its antecedent (filler). (2) Delete any empty elements introduced by step (1).

[8]Please refer to [2] for the details on how θ-roles are assigned.

2.3 Scrambling and Scope

Korean scrambling (as well as scrambling in other languages) is complicated further by examples such as these:

(7) (i) ne-nun [Minswu-ka nwukwu-lul coaha-nunci] a-ni
ne-nun Minswu-nom who-acc like-qm know-qm
'Do you know who Minswu likes?'

(ii) nwukwu$_i$-lul ne-nun minswu-ka t_i coaha-nunci a-ni

'Who do you know Minswu likes?' / 'Do you know who Minswu likes'[9]

In (7), (ii) has a different interpretation than (i). This is because scrambling interacts with scope interpretation. In [5], Lee claims that a scrambled wh-element (like "who" or "what") optionally reconstructs for scope interpretation. Reconstruction means that a scrambled NP optionally moves back to its unscrambled position.

Remember that at Logical Form (LF) interpretation (or, equivalently, scope interpretation at the FOPC level), the wh-word raises to the specifier position of the *nearest* CP (or Sentence.) The sentence is interpreted as a wh-question if a wh-word occupies the specifier position of the matrix (outer) CP. If a wh-word occupies the specifier position of the embedded (inner) CP, and the specifier position of the matrix (outer) CP is empty, the sentence is interpreted as a yes/no question.

In (7)(i), the wh-word in the embedded (inner) clause "nwukwu" raises to the nearest CP. and the whole sentence is interpreted as a yes/no question, as shown in (8):

(8) (i) [$_{CP}$ [$_{IP}$ ne-nun [$_{CP}$ [$_{IP}$ Minswu-ka *nwukwu* coaha-nunci]] a-ni]]

'The sentence before wh-raising'

(ii) [$_{CP}$ [$_{IP}$ ne-nun [$_{CP}$ *nwukwu*$_i$ [$_{IP}$ Minswu-ka t_i coaha-nunci]] a-ni]]

'*nwukwu* is raised to the specifier position of the nearest CP'

[9][5], example 160

In (8)(ii), wh-word occupies the specifier position of the embedded (inner) CP, and the specifier position of the matrix (outer) CP is empty, i.e., the sentence is interpreted as a yes/no question.

In (7)(ii), the whole sentence can be interpreted as a wh-question (as shown in (10)) as well as a yes/no question (as shown in (9).) For the yes/no interpretation (9), the scrambled wh-word reconstructs to its base position and then raises to the nearest CP.

(9) (i) $[_{CP}\ [_{IP}\ nwukwu_i$ ne-nun $[_{CP}\ [_{IP}$ Minswu-ka t_i coaha-nunci]] a-ni]]

'The sentence before wh-raising (*nwukwu* is scrambled to the front of the sentence.)'

(ii) $[_{CP}\ [_{IP}$ ne-nun $[_{CP}\ [_{IP}$ Minswu-ka *nwukwu* coaha-nunci]] a-ni]]

'Scrambled *nwukwu* reconstructed.'

(iii) $[_{CP}\ [_{IP}$ ne-nun $[_{CP}\ nwukwu_i\ [_{IP}$ Minswu-ka t_i coaha-nunci]] a-ni]]

'*nwukwu* raised to the nearest CP.'

In (9)(iii), a wh-word occupies the specifier position of the embedded (inner) CP, and the specifier position of the matrix (outer) CP is empty, i.e., the sentence is interpreted as a yes-no question.

For the wh-interpretation of (7)(ii), the scrambled wh-word raises to the nearest CP, without undergoing reconstruction.

(10) (i) $[_{CP}\ [_{IP}\ nwukwu$ ne-nun $[_{CP}\ [_{IP}$ Minswu-ka coaha-nunci]] a-ni]]

'The sentence before wh-raising (*nwukwu* is scrambled to the front of the sentence.)'

(ii) $[_{CP}\ nwukwu_i\ [_{IP}\ t_i$ ne-nun $[_{CP}\ [_{IP}$ Minswu-ka coaha-nunci]] a-ni]]

'Scrambled *nwukwu* is raised to the nearest CP without undergoing reconstruction.'

In (10)(ii), since a wh-word occupies the specifier position of the matrix (outer) CP. the sentence is interpreted as a wh-question. It is crucial to parse these examples correctly for a Korean Q/A system.

Here is how reconstruction is implemented: (1) Recurse down the parse tree and replace a scrambled wh-word with an empty element, and replace a trace (gap) of a wh-word with its antecedent (filler.) (2) Delete any empty elements introduced by step (1).

Reconstruction is incorporated into the "LF Movement" generator described in Figure 2-1. The original implementation of the "LF Movement" generator can be understood as two smaller generators serially linked (Figure 2-4):

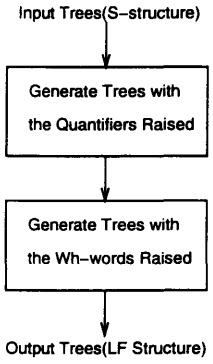

Figure 2-4: The Original Implementation of the LF Movement Generator

A new definition of the "LF Movement" generator is shown in Figure 2-5. (This definition will be revised in the following section).

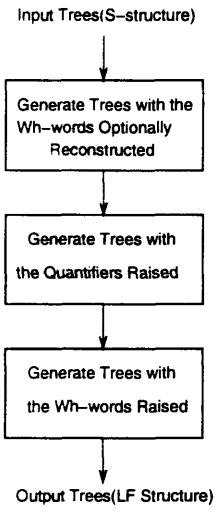

Figure 2-5: A New Implementation of the LF Movement Generator

2.3.1 Vacuous Wh-Chain Reconstruction

The implemented reconstruction algorithm must be more sophisticated if it is to avoid any redundant parses.[10] Here, redundant parses mean that two parses have the same scope interpretations at LF (typed first order predicate calculus) level. Consider this example:

(11) (i) **nwukwu**$_i$-lul [**pro** chinkwu]-ka t_i paypanhayss-ni
who-acc pro-gen friend-nom betrayed-Q
"**Who** did **his** friend betray"[11]

One possibility is to:

1. Reconstruct the chain ($nwukwu_i$,t_i), then

[10]The author would like to acknowledge Dr. Sandiway Fong's help with implementing the algorithms outlined in this subsection (Vacuous Wh-Chain Reconstruction) and the next subsection (Reconstruction for Subject Binding.)

[11][5], example 78b.

2. Raise "nwukwu" to the nearest CP at LF (or, equivalently, FOPC level.), as shown in (12)

(12) (i) $_{CP}$ [$_{IP}$ **nwukwu**$_i$ [**pro** chinkwu]-ka t_i paypanhayss-ni]]

'The sentence before wh-raising (**nwukwu** is scrambled to the front of the sentence.)'

(ii) $_{CP}$ [$_{IP}$ [**pro** chinkwu]-ka **nwukwu** paypanhayss-ni]]

'**nwukwu** is reconstructed.'

(iii) $_{CP}$ **nwukwu**$_i$ [$_{IP}$ [**pro** chinkwu]-ka t_i paypanhayss-ni]]

'**nwukwu** is raised to the nearest CP after reconstruction.'

Compare this with:

1. No reconstruction, then

2. Raise "nwukwu" from the original scrambled position to the nearest CP at LF, as shown in (13)

(13) (i) $_{CP}$ [$_{IP}$ **nwukwu**$_i$ [**pro** chinkwu]-ka t_i paypanhayss-ni]]

'The sentence before wh-raising (**nwukwu** is scrambled to the front of the sentence.)'

(ii) $_{CP}$ **nwukwu**$_i$ [$_{IP}$ t_i [**pro** chinkwu]-ka paypanhayss-ni]]

'Scrambled **nwukwu** is raised to the nearest CP without undergoing reconstruction.'

These two parses make no scope distinction, and are therefore redundant. The solution to this "vacuous wh-chain reconstruction" problem is to reconstruct only long distance wh-chains, i.e. only if there is some scope distinction to be derived. This eliminates the first possibility above.

The revised reconstruction algorithm is implemented as follows:

1. Mark each element of a long distance wh-chain, then

2. Replace the trace (gap) with the antecedent (filler), and replace the antecedent (filler) with a null element, for each member of the chain. Mark any null element so introduced for deletion.

3. Delete all elements marked for deletion.

Recall that all reconstruction is optional, so even if the scrambling is long distance, reconstruction may not occur.

2.3.2 Reconstruction for Subject Binding

Consider these examples:

(14) (i) [**caki** chinkwu]$_i$-eykey **nwukwuna**-ka t_i komin-ul thelenohnunta
self's friend-dat everyone-nom problem-acc tell.
"**Everyone** tells **his/her** friend problems"[12]

(ii) [**caki** uymwu]$_i$-lul **nwukwuna**-ka t_i chwungsilhi ihaynghayssta
self's duty-acc everyone-nom faithfully carried-out
"**Everyone** carried out **his/her** duty faithfully"[13]

When the quantifier "nwukwuna" is raised at LF, it produces a weak cross-over (WCO) violation.[14] Therefore, we need to avoid WCO violation by reconstructing the scrambled element which is bound by the subject (through the Subject Binding Generalization) before raising the quantifier.

Implementation:

1. Mark each element of a Subject Binding chain, then

2. Replace the trace (gap) with the antecedent (filler), and replace the antecedent (filler) with a null element, for each member of the chain. Mark any null element so introduced for deletion, then

[12][5], example 79b.
[13][5], example 80b.
[14]Weak crossover involves the coindexing of an empty category and a genitive inside an NP, as in **Who$_i$ does his$_i$ mother love e$_i$?* A WCO violation occurs when a wh-word or a quantifier raises from its D-structure position to the [spec, CP] and it "crosses over" a coindexed genitive inside an NP. The presence of a weak crossover makes the sentence unacceptable.

3. Delete all elements marked for deletion.

4. Check that the reconstructed tree satisfies conditions A[15], B[16], and C. If the tree violates any of these conditions, then do not reconstruct (The generator outputs the input tree unchanged.)

The tentative solution is to only allow Subject Binding reconstruction as a last resort for WCO violations.[17]

2.3.3 Implementation

Here is the algorithm that implements the ideas outlined above (Figure 2-6):

[15]Condition A states that an anaphor (e.g., "myself", "herself") must be bound within its governing category. The governing category of an NP is the smallest NP or Inflection Phrase (or, in standard notation, Sentence Phrase) containing that NP, its governor, and an "accessible" subject.

[16]Condition B states that a pronominal (e.g., "he", "they") must not be bound in its governing category.

[17]There is some overlap between the coverage of Subject Binding Generalization and reconstruction of the Subject Binding chain, but they have distinct functions, and both are needed. In the current implementation, Subject Binding Generalization is relevant when analyzing whether the sentence satisfies conditions A, B, and C in the S-structure. Subject Binding reconstruction is applied when the LF-structure is derived from the S-structure. So, without Subject Binding Generalization at the S-structure, reconstruction of the Subject Binding chain may never occur, since the parse tree may have been eliminated before reaching the LF-movement stage. Also, Subject Binding Generalization is relevant for both long and short distance scrambling, but reconstruction of the Subject Binding chain is only (optionally) applicable for long distance chains, as shown above.

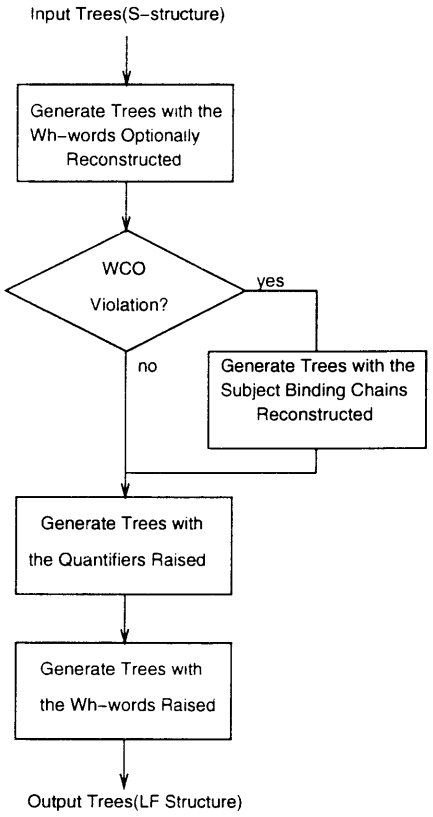

Figure 2-6: The Final Implementation of the LF Movement Generator

Chapter 3

Details

3.1 Scrambling Across More Than One Boundary

Lee states in [5], pg. 7, that "...it is not crystal clear whether scrambling across multiple clause boundaries is grammatical", citing as an example:

(15) (i) [na-nun [nwu-ka [sensayngnim-kkeyse Minho-lul pyenayhasinta-ko]
I-top who-nom teacher-nom Minho-acc like-excessively-comp
malhayss-nunci] kwungkumhata]
said-whether wonder
'I wonder who said that the teacher likes Minho excessively.'

(ii) *?Minho$_i$-lul [na-nun [nwu-ka [sensayngnim-kkeyse t_i pyenayhasinta-ko]
malhayss-nunci] kwungkumhata][1]

Lee states that the unacceptability of (15)(ii) is due to "reasons other than syntax" since not all Korean sentences with scrambling across more than one clause boundary is judged to be unacceptable. In **Pappi**, a parse is produced for (15)(ii). Notice that if IP (or equivalently, S) were defined as a bounding node, as in English, (15)(ii) would be ruled out by the Subjacency Condition. The Subjacency Condition states

[1] [5], example 10

that no single application of a movement rule may cross more than one bounding node. In the current implementation, only NP is stipulated to be a bounding node.[2]

3.2 A/A-bar Distinction

In [3], Fong claims that in Japanese, the landing site for a short distance scrambling is an A-position (argument position), e.g., object of a verb, while the "landing site" for a long distance scrambling is an A-bar position (non-argument position). In this paper, it is assumed that the landing site for both short and long distance scrambling is uniformly A-position in Korean, following Lee's analysis [5].

According to Lasnik and Uriagereka (as summarized in [2]):

- Let A be an empty NP in an A-position. Then:

  1. If B is in an A-position, then:
     A has feature `p(-)` if A and B do not have an independent θ-roles, i.e. are part of the same chain.

- Except for the case where A is a variable, the feature `a(+/-)` is freely assigned as follows:

  1. A may have feature `a(+)`
  2. A may have feature `a(-)` if the *pro*-Drop parameter is set
     (*pro*-Drop parameter is set for Korean)

Korean is in fact a *pro*-Drop language (like Spanish and Italian), which means that a verb argument, such as a subject or an object, can be empty. According to this definition, the trace of a long distance scrambled element would be assigned either `a(+)p(-)` or `a(-)p(-)` feature. If the trace is assigned `a(+)p(-)` feature,

[2]Lee claims that a subcategorized clause does not constitutes an island, while a non-subcategorized clause constitute a strong island. A subcategorized clause is a clause which is not an argument of the verb. A non-subcategorized clause is a clause which is adjoined, and is not an argument of the verb. An island is a constituent in which no dependency (or no dependency of a specified type) can have one end inside that constituent and the other end outside it. Since only subcategorized clauses are implemented in **Pappi**, IP is not defined as a bounding node.

Condition A is violated, since the trace is not bound within its governing category. (Condition A states that an NP with `a(+)` feature must be bound within its governing category. The governing category of an NP is the smallest NP or IP (or, equivalently, S) containing that NP, its governor, and an "accessible" subject.) If the trace is assigned `a(-)p(-)` feature, Condition C is violated, since the trace is bound by its antecedent. (Condition C states that an NP with `a(-)p(-)` must not be bound.)

Therefore, to allow long distance scrambling, the definition stated above must be modified:

- When the head of a chain binds its trace, and the chain is inter-clausal, the trace has feature `p(+)`. If the chain does not cross a clausal boundary, the trace has feature `p(-)`.

3.3 Anaphor Drop in Korean

Finally, consider this sentence:

(16) (i) Kim pancang-i **nwukwu**-eykey-na [**pro** iwus]-ul
Kim district chair-nom everyone-dat-uq pro-gen neighbor-acc
sokayhayssta
introduced

'The district chair Kim introduced **everyone** to **his** neighbor.'[3]

Lee claims that the empty element occupying the specifier position of "iwus" is the empty pronoun *pro*, which has features `a(-)p(+)`. Under this interpretation, **Pappi** would produce a Condition B violation for the above sentence. Condition B states that an NP marked with feature `p(+)` must not be bound in its governing category. Since "nwukwu" binds the empty element, above sentence violates Condition B, under Lee's interpretation.

To fix this problem, the `anaphorDrop` parameter was set in **Pappi** so that the empty element may be interpreted as a pure anaphor (with features `a(+)p(-)`.) The

[3] [5], example 75a

`proDrop` parameter is also set in Korean. so two interpretations are possible: the empty element as a *pro*, or as a pure anaphor. The *pro* interpretation is filtered out by the Condition B filter in the above example, so only the pure anaphor interpretation is possible.

Chapter 4

Comparisons with Other Systems

To implement the scrambling mechanism described above, less than 150 lines of Prolog code need to be added to the standard **Pappi** framework(See appendix A, B and C). Why is scrambling relatively easy to implement in this way? Essentially, **Pappi** can easily handle reconstruction since the code that encodes principles directly deals with sentence structures. In order to handle reconstruction, **Pappi** optionally moves the scrambled wh-elements back to their base position, and then raises them to the nearest CP at logical form interpretation. This is difficult in other systems proposed to handle Korean. First, consider Lin's PRINCIPAR ([1]). Since Lin's PRINCIPAR deals with the *description* of structures, it has difficulty dealing with reconstruction (unless the current design is drastically changed), since the messages it uses only pass *up* parse trees. In order for PRINCIPAR to handle reconstruction, the node representing a trace would have to know whether its antecedent is a wh-word, and decide if it should reconstruct. This would be difficult, since the trace cannot know its antecedent until a message from the trace and the message from the antecedent meet at a node which dominates both the trace and the antecedent. If the scrambled element is going to reconstruct, a message has to travel "down" the tree to the trace, which is not allowed in the current implementation of PRINCIPAR.[1]

Currently, neither PRINCIPAR nor the V-TAG formalism proposed to Korean

[1](Lin, p.c.) proposes implementing a filter after the structure has been built, to handle reconstruction, therefore abandoning the structure description idea for this part of the parse.

in [8] and [6] handle Binding Theory, e.g., Condition C. or Scope Interpretation. Both systems produce the different possible word orders. for both short and long distance scrambling. but neither systems capture the interaction between scrambling and binding, or the interaction between scrambling and scope interpretation.

Chapter 5

Parsing Time Analysis

Not surprisingly, scrambling does introduce additional computational complexity into parsing. An sample excerpt from our analysis is given below, where times are normalized to the unscrambled base time. It appears as though multiple scrambling beyond one clause results in the same nonlinear increases observed by Fong [2] for indexing.

Table 5.1: Parsing Time Analysis

	sentence	time, s.	ratio	comment
local multiple scrambling	1(a)	1.32	1	(no scrambling)
	1(b)	2.11	1.60	(one elem scrambled)
	1(c)	1.93	1.46	(one elem scrambled)
	1(d)	3.20	2.42	(two elem scrambled)
	1(e)	2.46	1.86	(one elem scrambled)
	1(f)	3.32	2.52	(two elem scrambled)
long distance multiple scrambling	3(a)	6.61	1	(no scrambling)
	3(b)	8.90	1.35	(one elem scrambled)
	3(c)	15.92	2.41	(two elem scrambled)

Chapter 6

Conclusions

This paper describes how one of the most current linguistic analyses of Korean scrambling can be readily incorporated into an existing parser. The parser can correctly handle 26 out of all 29 sets of examples in chapters 2.2 (excluding 2.2.6 on parasitic gaps) and 3.2 of [5]. The interaction between scrambling and other components of the grammar is easily accommodated, just as described by Lee. The approach outlined in this paper is compared with other approaches to scrambling, and surpasses them in coverage. The directness with which Lee's linguistic theory can be modeled demonstrates the value of using a "transparent" principles and parameters approach. We can simply use the theoretical assumptions that Lee makes and then test her results using the wide range of scrambling data she exhibits.

Appendix A

Code Implementing Subject Binding Generalization

```
binds(A,B,CF) :-                % redefinition of "binds"
        (binds_in_DS(A,B,CF) ; c_commands(A,B,CF)),
        coindexed(A,B).

binds_in_DS(A,B,SS) :-
        c_commands_in_DS(A,B,SS),
        subject(A),
        make B have_feature beta_marked.

% notice that the subject never scrambles....

subject(A) :- A has_feature theta(agent),
        make A have_feature subject.

%% A c-commands B in D-struct

c_commands_in_DS(A,B,S) :-
        recoverDsRecur(D,S),
```

```
        D has_constituents Cs,
        member(A1,C,Cs),
        transparent(A1,A),
        dominates_mine(C,B).
c_commands_in_DS(A,B,S) :-
        S has_constituent C,
        c_commands_in_DS(A,B,C).


recoverDsRecur([DS,DLeft,DRight], [SS,SLeft,SRight]) :-
        recoverDsElement([DS],[SS]),
        [DS] has_feature _       % DS is not empty
        -> recoverDsRecur(DLeft, SLeft),
           recoverDsRecur(DRight, SRight)
       ;   true.
recoverDsRecur(DS,[SS,Word]) :-
       recoverDsElement(DS,[SS,Word]).
recoverDsRecur(DS,[SS]) :-
       recoverDsElement(DS,[SS]).


dominates_mine([C$_$[Fs1|_]--_],[C$_$[Fs2|_]--_]) :-
        sameCategory(Fs1,Fs2),
        !.
dominates_mine([C$_$[Fs1|_]--_,Word],[C$_$[Fs2|_]--_,Word]) :-
        sameCategory(Fs1,Fs2),
        !.
dominates_mine(A,B) :-
        A has_feature _,
        A has_constituent C,
```

```
dominates_mine(C,B).
```

Appendix B

Code Implementing Reconstruction

```
lfMovement(SS,LF) :-
        reconstructWhChain(SS1,SS),          % optionality built in
        (\+ canReanalyze(SS1)                % last resort
        -> reconstructForSB(SS2,SS1)
        ;  SS2 = SS1 ),
        qr(SS2,SS3),                         % quantifier raising
        moveWh(SS3,LF).                      % wh-word raising

canReanalyze(SS) :-
        qr(SS,SS1),
        moveWh(SS1,SS2),
        licenseOpVars(SS2),
        reanalyzeBoundProforms(SS2).

reconstructWhChain(SSp,SS) :-
        markLDWhChain(SS),
        reconstructOne(SS1,SS),
        reconstruct(SS2,SS1),
```

```
        delECs(SS2,SSp).
reconstructWhChain(SS,SS).                          % optional

reconstruct(SSp,SS) :-
        reconstructOne(SS1,SS),
        !,
        reconstruct(SSp,SS1).
reconstruct(X,X).

reconstructOne(DS,SS) :-
        (reanalyze X from SS given_by recChainElement(X)
               as Y from DS given_by reconstructPhrase(X,Y)),
        !,
        DS \== SS.

recChainElement(X) :-
        maximalProj(X), partOfChain(X), X has_feature reconstruct.

reconstructPhrase(X,Y) :- recoverDsElement(Y,X), addFeature(del,Y) if null(Y).

delECs(XP,YP) :- delOneEC(XP,ZP) -> delECs(ZP,YP) ;  YP = XP.

delOneEC(XP,YP) :-
        XP has_constituents L,
        ((pick(Null,L,[YP]),
          ec(Null),
          Null has_feature del)
        -> true
        ;  delEC1(L,Lp),
           YP has_constituents Lp,
```

```
            YP shares_category_and_features_with XP).

delEC1(L,Lp) :-
        append(Left,[X|Right],L),
        delOneEC(X,Y),
        append1(Left,[Y|Right],Lp).

% Reconstruct for Subject Binding

reconstructForSB(DS,SS) :-
        markSBChain(SS),
        reconstructOne(SS1,SS),
        reconstruct(SS2,SS1),
        delECs(SS2,DS),
        conditionA(DS),                         % make sure reconstructed
        conditionB(DS),                         % structure satisfies BT
        conditionC(DS).

moveOneWh(CP,CPp,WhChain,noblock) :- % raise one wh from within IP to this CP
        cat(CP,c2),
        CP has_feature q,
        IP complement_of CP,
        extract whLF(_) from IP producing IPp and WhChain,
        WhChain = [Wh,Trace],
        \+ intermediateCP(IPp,Trace),
        Trace has_feature ec(wh),
        addToCP(Wh,IPp,CP,CPp),
        addFeature(blocked,CPp).
moveOneWh(XP,XPp,WhChain,Blocked) :-
        XP has_constituents L,
```

```
        append(Left,[LeadsToWh|Right],L),
        propagateBlocking(XP,Blocked,Blockedp),
        moveOneWh(LeadsToWh,Raised,WhChain,Blockedp),
        \+ LeadsToWh = Raised,
        append(Left,[Raised|Right],Lp),
        XPp has_constituents Lp,
        XP shares_category_and_features_with XPp.

intermediateCP(IP,Trace) :-
        IP dominates CP,
        cat(CP,c2),
        CP has_feature q,
        CP dominates Trace.
```

Appendix C

The Rest of the Periphery File (Excluding the Code Listed in Appendix A and B.)

```
%%%  -*- Package: PROLOG-USER; Mode: PROLOG -*-

%%% PERIPHERY FOR KOREAN
%%%
%%% From peripheryJapanese.pl, modifications by Frank Cho and Sandiway Fong.
%%%
%%% Language-particular operations + kludgey stuff
%%% 1. case agreement
%%% 2. constrained scrambling
%%%

%%% S-STRUCTURE GRAMMAR ADDITIONS

% Experimental feature pushing
pushFeature(morphC(_)).
```

```
rule ecNP      -> [np(NP)] st ec(NP).
rule opC2$c2   -> [ecNP,c1].

rule head_adjoined _ adjoins_to_the left.    % in head movement

% Pushed features: Will be automatically generated...

rule dObjectNP  -> [np(NP)] st \+ C==nom if NP has_feature morphC(C).
rule ioObjectNP  -> [np(NP)] st C==dat if NP has_feature morphC(C).
rule overtONP -> [overtNP(NP)] st (C==acc;C==dat) if  NP has_feature morphC(C).
rule objectNP -> [np(NP)] st (C==acc;C==dat) if  NP has_feature morphC(C).
rule subjectNP -> [np(NP)] st \+ (C==acc ; C==dat) if NP has_feature morphC(C).
rule npSubjectNP -> [np(NP)] st C==gen if NP has_feature morphC(C).

adjunction rule vp -> [overtONP,vp].    % object scrambling (VP-int)
adjunction rule i2 -> [overtONP,i2].    % no intermediate traces
%adjunction rule vp -> [subjectNP,vp].    % hack
%adjunction rule i2 -> [subjectNP,i2].    % hack
adjunction rule i2 -> [pp,i2].           % hack by fscho

% Base adjunction
adjunction rule np -> [overtNP,nq].    % freely adjoin NQ to NP
adjunction rule np -> [nq,np].
%adjunction rule np -> [pp,np] st lexicalProperty(pp,conj).

rhs [n1(N1)]        add_goals [noSubject(N1)].  % sandiway: [NP pro N1] vs. [NP N1]
rhs [overtONP(NP),vp] add_goals [aPos(NP)].    % scramble object to A-pos
rhs [overtONP(NP),i2] add_goals [aPos(NP)].    % A-pos (tentatively)

rhs [vp(VP),v] add_goals [\+ adjoined(VP)].    % eliminate unnecessary
```

```
  % non-determinism
% NQ NP Agreement
rhs [overtNP(NP),nq(NQ)] add_goals [agreeNPNQ(NP,NQ)]. % eliminate non-det.
rhs [nq(NQ),np(NP)] add_goals [agreeNPNQ(NP,NQ)].

% Scrambling
lhs overtONP add_goals [pushReq(es(i),es(o))].
lhs dObjectNP & rhs [np(X)] add_goals [cReq(X,es(i),es(o))].
lhs ioObjectNP & rhs [np] add_goals [cReq(es(i),es(o))].
lhs subjectNP & rhs [np(X)] add_goals [ldReq(X,es(i))].
lhs leftvgridcsr1stnp add_goals [oneReq(es(i))].
lhs v0 add_goals [zeroReq(es(i))].

rhs [det,n1] add_inherit plus(2,[1,[wh(_),op(_)]]).

% rhs [pp,vp] replace_rhs [coPP,vp].
rhs [c2,relClNP] replace_rhs [opC2,relClNP].

% Experimental feature pushing, again...
rhs [np,vgrid] replace_rhs [dObjectNP,vgrid].   % opt. direct object
rhs [np,v1] replace_rhs [ioObjectNP,v1].   % opt. indirect object
rhs [np,i1] replace_rhs [subjectNP,i1].   % opt. subject
rhs [np,pgrid] replace_rhs [overtNP,pgrid].   % disallow post-pos stranding
rhs [np,n1] replace_rhs [npSubjectNP,n1].   % genitive Case

% null C not permitted for argument CPs
lhs v1gridcsr1stc2 & rhs anywhere c2(X) app_goals [(\+ nullComp(X))].

left_bracket c2 substitute openReq for open.
```

```
% scrambling hack.... overgenerates.
%rule c2 with Features -> [] st nullFeatures(Features).        % hack
%lhs c2 add_goals [zeroReq(es(i))].
%%lhs c2 add_goals [pushReq(es(i),es(o))].
%adjunction rule i2 -> [c2,i2].          % hack by fscho

%%% S-STRUCTURE GRAMMAR DELETIONS

%% kind of redundant, will not be needed in next version

block rule adv -> [adv(Adv)] st maybeSubcategorized(adv,Adv).

%%% OTHER LANGUAGE-SPECIFIC AREAS

%%% EMPTY COMP

emptyCompFeatures(Fs) :- mkFs([wh(-)],Fs).

%% Null Comp C2
nullComp(CP) :-
C head_of CP,
ec(C).

%%% Move-Alpha (D-structure to S-structure)

moves(CF,np) :- cat(CF,np).
moves(CF,np) :- cat(CF,cp).
moves(CF,c2) :- cat(CF,np).
moves(CF,c2) :- cat(CF,cp).
```

```
noSubject(X) :- \+ (X has_feature grid(Ext,_), Ext \== []).

% compatibleCase(AssignedCase,MorphologicallyRealizedCase)

compatibleCase(X,X).
compatibleCase(_,topic).    % deal with topicalization later

% Case Transmission: Need it for scrambling (complement to adjunct)
% NB. need to do [NP NQ NP-t], despite extraction, NQ-NP is overt

caseTransmission(Hd,NP,Case) :-
baseTrace(NP),
headOfChain(Head,NP),
Head has_feature adjunct,    % scrambling
NP has_feature compl,
assignSCase(Hd,Case,Head),
NP has_feature case(Case) if \+ ec(NP).    % [NP NQ NP-t]

% Indicate all non-adjunct Case are realized using overt markers.

realizedAsMarker(X) :- X \== obq.

caseRealizationMode(_NP,morphC).

% NQ NP agreement

agreeNPNQ(NP,NQ) :-
NQ has_feature classifier(Class),
((\+ ec(NP) ; NP has_feature class(_))
-> agreeClassifier(NP,Class)
```

```
;  NP has_feature ec(trace),    % force trace
   transmitViaChain([],[goal(agreeClassifier1(X,Class),X)],NP)).

agreeClassifier1(NP,Class) :- agreeClassifier(NP,Class).

agreeClassifier(NP,Class1) :-
NP has_feature class(Class)
-> Class = Class1
;  Class1 = default.

%% Chain Formation conditions

chainLinkConditions(Head,Trace,_,UpPath,DownPath) :-
\+ vacuousScrambling(UpPath,DownPath)
if Trace has_feature_set [apos,adjunct],  % scrambling
longDistABarPos(Head,UpPath)
if Head has_feature_set [apos,adjunct].

%chainLinkConditions(Head,Trace,_,UpPath,DownPath) :-
% \+ vacuousScrambling(UpPath,DownPath)
% if Trace has_feature_set [apos,adjunct].  % scrambling

vacuousScrambling([],_).    % no topmost segment crossed
vacuousScrambling(_,Down) :- \+ Down == [].

% Long Distance scrambling is A-bar
%
% Modification:
%  1. Well, perhaps only to IP-adjunction sites...
%     See [98b] in lee.xpl
```

```
%longDistABarPos(Head,UpPath) :-
% addFeature(goal(apos,fail),Head)
%     if (in(c2,UpPath), last(UpPath,i1)). % inter-clausal

longDistABarPos(Head,UpPath) :- true.

%% SCRAMBLING
%%
%% Must prevent vacuous scrambling

shiftRequest(n,es).

% request carrier r(State)
% State = Var or 1

%% pushReq(ES,ES') start new req state 0
%% shiftReq(ES) all requests state 0 -> 1
%% cReq(ES,ES') ticks off a state 1 req
%% cReq(X,ES,ES') X must be ec if req found
%% openReq(ES,ES')  put in place of open, barf if state 0 req found

% initiates a request
pushReq(ES,[r(_)|ES]) :- kReq(ES).

% handles r([X])
kReq([X|ES]) :-
open(X)
-> true
;  (functor(X,r,_)
```

```
   -> kReq1(ES)
   ;  kReq(ES)).

% fails if we get to r(_) before an open
kReq1([X|ES]) :- open(X) -> true ;  \+ functor(X,r,_), kReq1(ES).

% change state of all open requests
% handles r([X])

shiftReq([X|ES]) :-
open(X)
-> true
;  ((X = r(1) ; X = r([1]))   % state <- 1
   -> shiftReq(ES)
   ;  shiftReq(ES)).

ldReq(Item,ES) :- ec(Item) -> ldReq(ES) ; true.

ldReq([X|ES]) :-
open(X)
-> true
;  (X = r(V)
   -> (var(V) -> V = [_] ; true),
      shiftReq(ES)
   ;  shiftReq(ES)).

%  consume one shifted request

cReq(ES,ESp) :-
ES = [X|ES1],
```

```
 (open(X)
 -> ESp = ES
 ;  (X = r(S)
    -> (S == 1    % consume
       -> ESp = ES1
       ;  ESp = [X|ESp1],
          cReq(ES1,ESp1))
    ;  ESp = [X|ESp1],
       cReq(ES1,ESp1))).


% obligatory consume shifted request

cReq(Item,ES,ESp) :-
ES = [X|ES1],
(open(X)
-> ESp = ES
;  (X = r(S)
   -> (S == 1    % shifted
      -> withEmpty(Item),
         ESp = ES1
      ;  ESp = [X|ESp1],
         cReq(Item,ES1,ESp1))
   ;  ESp = [X|ESp1],
      cReq(Item,ES1,ESp1))).

withEmpty(X) :- ec(X) -> true ; adjoined(X,_,X1), withEmpty(X1).

% non-local request propagation
% ES = [...Rs...]
```

```
% ES' = [..Rs...,open,...]
% Translates r([1]) -> r(1)

openReq(ES,ESp) :-
nlReq1(ES,ES1,Rs),
append1(Rs,ES2,ESp),
open(ES1,ES2).

% separates local requests Rs leaving ES'
nlReq1([],[],[]).
nlReq1([X|ES],ESp,Rs) :-
open(X)
-> ESp = [X|ES],
   Rs = []
;  (X = r(S)
   -> (S == 1   % already shifted
      -> Rs = [X|Rsp],
         nlReq1(ES,ESp,Rsp)
      ;  S == [1],   % shift, xform r([1])->r(1)
         Rs = [r(1)|Rsp],
         nlReq1(ES,ESp,Rsp))
   ;  ESp = [X|ESp1],
      nlReq1(ES,ESp1,Rs)).

% <= 1 state 1 req, no state 0 req
oneReq([X|ES]) :-
open(X)
-> true
;  (X = r(S)
   -> S == 1,
```

```
      zeroReq(ES)
   ;  oneReq(ES)).

% no reqs of any state allowed

zeroReq([X|ES]) :- open(X) -> true ;  \+ functor(X,r,_), zeroReq(ES).

%%% LEXICON SUPPORT

% Priority correct?

externalRolesForNi(X,Y) :-
vpAllowExtL([goal,source],X)
-> Y = goal
;  unsaturatedExtRole(X,agent),
   Y = agent.

%%% All NP's with 1st person feature must be coindexed

%coindex(NP1,NP2) :-firstPersons(NP1,NP2).

%firstPersons(NP1,NP2) :-
% cat(NP1,np),
% cat(NP2,np),
% NP1 has_feature agr(Feature1),
% NP2 has_feature agr(Feature2),
% member(1,Feature1),
% member(1,Feature2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%% To allow a PP to scramble......

adjunctLicensed(vp,VP,Adjunct) :- % (1) VP[aux] & [-aux]
VP has_feature grid(_,_),
(cat(Adjunct,np)
->  true % scrambling
;   Adjunct has_feature predicate(_)).
adjunctLicensed(vp,VP,_) :-
VP has_feature aux,
inheritsFeature(VP,grid([_Role],_)).

adjunctLicensed(ap,_AP,Adjunct) :- cat(Adjunct,c2).

adjunctLicensed(i2,_IP,Adjunct) :-
( cat(Adjunct,np) ; cat(Adjunct,pp) )   % hack by fscho
-> true % scrambling
;  clause(Adjunct). % clausal extraposition

adjunctLicensed(np,NP,Adjunct) :-
cat(Adjunct,nq)
-> true
;  relClauseConfig(NP,LowerNP),
   \+ relClauseConfig(LowerNP,_),
   cat(Spec,np),
   Spec specifier_of Adjunct,
   operator(Spec),
   agreeAGR(Spec,LowerNP),
   link(Spec,LowerNP).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% redefinition of Functional Determination
%% When the head of the chain binds the trace, and the chain is inter-clausal,
%% the trace is p(+).  If the chain does not cross the clausal boundary,
%% the trace is p(-).

setABoundFs(Binder,Bindee) :-
partOfSameChain(Binder,Bindee) % binder and bindee share O-roles
        -> ((upPath(Bindee,UpPath),
    in(c2,UpPath))                          %inter-clausal
   -> Bindee has_feature p(+)               % hack by fscho
   ;  Bindee has_feature p(-)),
   Bindee has_feature a(+) if \+ proDrop
;  mostlyPro(Bindee).
```

Appendix D

The Parameters File for Korean

```
%%%  -*- Mode: PROLOG; Package: PROLOG-USER -*-

%%% KOREAN PARAMETER SETTINGS
%%%
%%% From parametersJapanese.pl, modifications by Frank Cho and Sandiway Fong
%%%

%%% REFERENCES
%%% no P utilities

%% X-Bar Parameters

specInitial.
specFinal :- \+ specInitial.

headInitial(X) :- \+ headFinal(X).
headFinal(_).

agr(strong).
```

```
%% V2 Parameters
% C is not available as adjunction site
% Empty C is null C

%% Subjacency Bounding Nodes

boundingNode(i2).
boundingNode(np).

%% Case Adjacency Parameter

no caseAdjacency.

%% Wh In Syntax Parameter

no whInSyntax.

%% Pro-Drop

proDrop.

%% Negation

negationMoves.

%% No Stranding

no allowStranding.

%% Allow null Case markers for empty Chains
```

```
nullCasemarkers.

%% null Anaphor

anaphorDrop.

%% Clitics

no clitic(_).

%% License object pro parameter

no licenseObjectPro.
```

Bibliography

[1] Bonnie J. Dorr, Dekang Lin, Jye hoon Lee, and Sungki Suh. Efficient parsing for korean and english: A parameterized message passing approach. *Computational Linguistics*, 1995.

[2] Sandiway Fong. *Computational Properties of Principle-Based Grammatical Theories.* PhD thesis, MIT, Cambridge, MA 02139, 1991.

[3] Sandiway Fong. Towards a proper linguistic and computational treatment of scrambling: An analysis of japanese. In *Proceedings of COLING-94*, 1994.

[4] Robert Frank, Young-Suk Lee, and Owen Rambow. Scrambling as non-operator movement and the special status of subjects. In *Proceedings of the Third Leiden Conference for Junior Linguists*, 1992.

[5] Young-Suk Lee. *Scrambling as Case-Driven Obligatory Movement.* PhD thesis, University of Pennsylvania, Philadelphia, PA 19104-6228, 1994.

[6] Hyun Seok Park. Korean grammar using tags. Master's thesis, University of Pennsylvania, Philadelphia, PA, 1994.

[7] Owen Rambow. *Formal and Computational Aspects of Natural Language Syntax.* PhD thesis, University of Pennsylvania, 3401 Walnut Street, Suite 400C, Philadelphia, PA 19104-6228, 1994.

[8] Owen Rambow and Young-Suk Lee. Word order variation and tree-adjoining grammar. *Computational Intelligence*, 10(4):386–400, 1994.